Identifying outdated requirements based on source code changes

Ben Charrada, Eya ; Koziolek, Anne ; Glinz, Martin

Abstract: Keeping requirements specifications up-to-date when systems evolve is a manual and expensive task. Software engineers have to go through the whole requirements document and look for the requirements that are affected by a change. Consequently, engineers usually apply changes to the implementation directly and leave requirements unchanged. In this paper, we propose an approach for automatically detecting outdated requirements based on changes in the code. Our approach first identifies the changes in the code that are likely to affect requirements. Then it extracts a set of keywords describing the changes. These keywords are traced to the requirements specification, using an existing automated traceability tool, to identify affected requirements. Automatically identifying outdated requirements reduces the effort and time needed for the maintenance of requirements specifications significantly and thus helps preserve the knowledge contained in them. We evaluated our approach in a case study where we analyzed two consecutive source code versions and were able to detect 12 requirements-related changes out of 14 with a precision of 79%. Then we traced a set of keywords we extracted from these changes to the requirements specification. In comparison to simply tracing changed classes to requirements, we got better results in most cases.

DOI: https://doi.org/10.1109/RE.2012.6345840

Identifying Outdated Requirements Based on Source Code Changes

Eya Ben Charrada, Anne Koziolek, Martin Glinz
Departement of Informatics, University of Zurich, Switzerland
{*charrada, koziolek, glinz*}*@ifi.uzh.ch*

***Abstract*—Keeping requirements specifications up-to-date when systems evolve is a manual and expensive task. Software engineers have to go through the whole requirements document and look for the requirements that are affected by a change. Consequently, engineers usually apply changes to the implementation directly and leave requirements unchanged.**

In this paper, we propose an approach for automatically detecting outdated requirements based on changes in the code. Our approach first identifies the changes in the code that are likely to affect requirements. Then it extracts a set of keywords describing the changes. These keywords are traced to the requirements specification, using an existing automated traceability tool, to identify affected requirements.

Automatically identifying outdated requirements reduces the effort and time needed for the maintenance of requirements specifications significantly and thus helps preserve the knowledge contained in them.

We evaluated our approach in a case study where we analyzed two consecutive source code versions and were able to detect 12 requirements-related changes out of 14 with a precision of 79%. Then we traced a set of keywords we extracted from these changes to the requirements specification. In comparison to simply tracing changed classes to requirements, we got better results in most cases.

***Keywords*-requirements update, traceability, source code changes, software evolution**

I. Introduction

Requirements specifications are used by engineers for several maintenance-related tasks [1], such as comprehending programs, getting the rationale behind the implementation, identifying critical parts in the system, and discussing changes with stakeholders. Therefore, losing the knowledge contained in the requirements specification hinders the maintainability of software systems and limits their capacity for evolution. Nevertheless, the requirements specification is often not updated when the software evolves [2] [3] [4], because updating the requirements document, which might include hundreds or thousands of pages written in natural language, is still a manual task that requires a lot of time and effort. Therefore, engineers usually choose to apply changes to the implementation only and leave the requirements document unchanged, as observed by e.g. Lethbridge et al. [3]. As a result, the specification becomes outdated and looses its value.

The existing approaches for keeping requirements up-to-date can be classified into two categories: *Normative approaches* require developers to update requirements first, before any code changes are made [5]. However, these approaches suffer from large manual effort to update both artefacts. *Trace generation approaches* such as [6] aim at generating traces between requirements specification and code in order to support developers when analyzing the impact of a change in one artefact on the other. However, these approaches aim at generating all traces and do not consider the context of a specific change.

The work we present in this paper aims at supporting the update of requirements specifications when software systems evolve. We assume a situation that is frequently encountered in practice: An engineer modifies an existing system by making changes to the source code. No traces between code and requirements exist. The engineer knows that he should update also the requirements specification. However, under the usual time pressure, he will only do this if it can be done with little additional effort. Our approach provides help at this point: By analyzing the changes in the source code, we semi-automatically identify the requirements affected by these changes and need to be updated, hence.

Our approach is composed of two main components. The first component is a code differencing technique that focuses on identifying source code changes that are likely to affect requirements. The technique was built based on an exploratory case study, where we made several observations of how we can differentiate between such requirements-related changes in code and changes that are refactorings or bug fixes. The second component of the approach is a technique for extracting relevant keywords describing the identified change and its context. The keywords are extracted from the name of the changed elements in the code, their documentation and their call graph. These keywords can then be traced to requirements using any automated traceability tool in order to identify the requirements that are likely to be impacted by the change.

To validate our approach, we applied it to a case study of a health care system. In the first part of the validation we assessed the effectiveness of our approach for identifying requirements-related changes. In the second part, we evaluate the effectiveness of using change-related keywords for tracing to requirements instead of tracing the class directly. In the change identification part, our tool succeeded to detect 12 of the 14 requirements-related changes, while extracting considerably fewer changed classes than the normal Eclipse comparator (33 with our approach against 91 with Eclipse), thus providing less irrelevant information. In the second part,

our tool was able to provide a better ranking of potentially outdated requirements than a class-based tracing approach.

Our approach is expected to help the developer first to identify the changes in the code that are likely to affect the external behaviour of the system and then to find the requirements that are related to them. The approach can either be used in a fully automated way, where the changes are directly traced to requirements or in a semi-automated way, where the user can filter out manually the changes that he thinks are not relevant before running the tracing. Even in the semi-automated configuration, the manual effort required from the maintainer is small. Automatically identifying outdated requirements will make the life of the maintainer easier as it will reduce the time and effort needed for performing the update. Thus it should also encourage engineers to maintain the requirements after each code release.

The contribution of this paper is a novel approach for semi-automatically identifying outdated requirements when software systems evolve. This work mainly targets functional requirements. The approach contains two novel features: First, we identify possibly requirements-related changes in source code based on observations how requirements-related changes differ from refactorings and bug-fixes. Second, we propose to extract keywords for tracing only from the changed elements and their context, such as call hierarchy and containing code elements. Furthermore, we provide a prototypical implementation and validate it in a case study.

The paper is organized as follows. In Section II, we describe an exploratory case study to characterize source code changes that likely affect functional requirements and to derive heuristics to identify such changes. Section III describes our approach to automatically detect outdated requirements based on changes in the code. Section IV presents the validation of our approach in the case study. Finally, Section V discusses related work, Section VI highlights issues for future research, and Section VII concludes.

II. Exploratory Study: Identifying Relations Between Changes in Code and Changes in the System External Behaviour

Functional requirements usually describe the external behaviour of the system, therefore in this work we will assume that the changes affecting the external behaviour of the system are also affecting the functional requirements. To know which types of changes in code are likely to affect the external behaviour of the system, we conducted a small exploratory study using a real software project, namely an open source project for a barcode reader called ZXing[1]. The research question of this study is

> RQ1: What heuristics can be used to identify source code changes that likely affect the external behaviour of the system?

[1] http://code.google.com/p/zxing/

We compared two versions of the source code and made observations about how to differentiate between changes that are likely to be related to changes in system behaviour and changes that are refactorings or bug fixes. In our exploratory study, we went through all the changes between the versions 1.6 and 1.7 of ZXing manually and studied how requirements-related changes differ from refactorings and bugfixes. For some randomly selected packages, we studied the changes in detail and counted the frequencies. In this section, we present the six observations we made and what heuristics for identifying relevant changes we can derive from them.

Observation 1: Changes in methods bodies are in most cases related to refactoring and/or bug fixes: Changes in methods bodies are among the most frequent changes that are applied to the code. However, they are not the most important ones in terms of affecting the external behaviour of the system. In fact, most of the changes observed in the body of methods in the explored project where minor changes that relate to either refactorings or bug fixes. We also noticed that the few changes in methods bodies that related to additions or extensions of features to the system came along with additions of new elements (e.g. new classes, methods or fields). For example, in the packages that we chose to confirm the observation, we identified 33 changes in methods bodies. 23 of these changes were due to refactorings and bug fixes (the majority, 19, being refactorings) and 6 changes were related to the additions of new features. For the other 4 changes we could not guess what was the intent of the developer behind it. All the changes related to feature extension came along with additions of new elements in the code. Based on these observations, we derive the heuristic to ignore the changes in methods bodies.

Observation 2: Additions of new elements (classes; methods; package; fields) are usually related to the addition or extension of features: We noticed that extension and addition of features are in most cases implemented through an addition of new elements in the code, where the names of these added elements usually reflect the implemented feature. It is important to note that there were some cases where the added element only relates to some implementation details. Therefore it is wrong to assume that *all* additions are extensions. However, we still can derive the heuristic that additions of new elements (additions of packages, classes, methods and/or fields) likely affect the external behaviour of the system.

Observation 3: Additions and removals of elements having similar names are usually rename operations: When using normal differencing tools, renames are detected as an addition and a removal of two different elements. This can be very misleading as addition of elements is likely to relate to feature extension while renames are simple refactorings. When exploring the ZXing project, we noticed that in many cases, the new name is very similar to the old one (e.g. the

field PDF417 was renamed to PDF_417). Therefore renames could be identified by computing the similarity between the name of the deleted and the name of the added one.

Observation 4: Changes in methods signature are usually related to refactoring: Changes in methods signatures (other than renaming the method) were among the frequent changes that we observed when exploring the ZXing project and were in most cases related to refactoring. These changes can affect the visibility of the method (public, private, etc.), its return type (e.g. int, boolean, object...) and/or the parameters of the methods (e.g. the type of the parameters). In the packages we used for confirming the observation, all of the signature changes were due to refactoring. We derive the heuristic to ignore such signature changes.

Observation 5: Changes in private elements can affect the external behaviour of the system: When starting our exploratory study, we were expecting to find that changes in public elements are likely to affect the external behaviour of the system while private elements will only relate to implementation details. However, this was not the case in the ZXing project, as there were many changes in private elements that affected the external behaviour of the system. Therefore, we include changes in private elements when looking for changes affecting the system behaviour.

Observation 6: Additions of several methods having the same name are usually related to the same feature: In many cases, we noticed that several methods having exactly the same name but different parameters where added to a class. In almost all cases, these methods related to the same feature and had similar behaviour. Therefore we derive the heuristic to consider and analyse only one of the added methods instead of considering them all.

III. Approach for Identifying Outdated Requirements

This section presents our approach for identifying outdated requirements. The approach consists of 3 steps (see Figure 1). First we identify the changes that are likely to affect the external behaviour (Section III-A) based on the observations and heuristics presented in the previous section. Then, for each change we extract a list of keywords from the names of the changed elements, their documentation and their call hierarchy (Section III-B). Finally we trace the extracted keywords to the requirements specification in order to identify outdated ones (Section III-C). To do the tracing automatically, we use an existing traceability tool based on information retrieval.

A. Identifying Relevant Changes

The first step of our approach is to detect the differences between the two versions of the source code and identify the relevant changes, i.e. changes that are likely to relate to a change in the external behaviour in the system and thus to relate to changes in requirements. Ideally refactorings, bug fixes and changes in the code documentation should not be detected by the approach. For achieving this goal, we developed a simple comparison technique which builds upon the heuristics presented in Section II. Our approach is targeting code written in object-oriented programming languages.

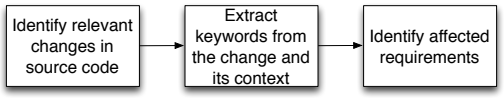

Figure 1. Approach for identifying requirements affected by change

We suppose that the code is composed of the following elements: packages, classes, methods and fields. Our comparing strategy consists in focusing on only two types of change: addition and removal of elements. First, we compare the packages in both versions and detect those that have been added or removed. The comparing is done based on the name only: a package is considered as added to the new version (respectively removed from the old version) if there is no other package in the old version (respectively the new version) that has the same name. Second, we go through each of the packages that appear in both versions of the code, and compare the classes it contains. Here again we do the comparing based on the class name to detect added and removed classes. Third, we go through each class that appears in both versions and compare the methods and fields it contains. By detecting all added elements, this approach is detecting the main code changes that relate to feature addition/extension (observation 2). At the same time, the approach ignores many of the changes that are usually related to refactoring such as changes in method bodies (observation 1), changes in element signatures (observation 4) and changes in documentation.

As our comparing technique is based on element names only, renaming is detected as a simultaneous addition and removal of two elements. To filter out renames, we compare the names and the call hierarchy (for classes and methods) of the added and the removed elements. If the added and deleted elements belong to the same parent element (e.g. two fields belong to the same class) and if they have similar names, then the change is considered as a rename (observation 3) and is ignored by our approach. In the case of methods and classes, we also explore the call hierarchy of the elements: if the added and the deleted element have the same call hierarchy, then it is a rename. The similarity between the names of elements can be calculated in several ways such as using the Levenshtein distance [7].

The output of this step is a set of source code elements that have been added and deleted and which are supposed to be requirements-related.

Table I
ELEMENTS USED FOR EXTRACTING KEYWORDS FOR EACH TYPE OF CHANGE

Changed Element	Names	Documentation	Call Hier.
Package	package, sub-classes	none	No
Class	class, sub-methods, sub-fields	class	Yes
Method	method, parent class	method, parent class	Yes
Field	field, parent class	parent class	No

B. Extracting Keywords

In this part, we extract a list of keywords describing the change. We consider three sources of keywords (see Table I): (1) the names of elements related to a change in the code (more details are given in the next paragraphs), (2) the documentation of elements and (3) their call hierarchy. The list is then reduced by filtering out irrelevant keywords and by grouping keywords. We detail each of the steps in the reminder of this section.

1) Names of the Change-Related Elements: The name of an element in the source code usually reflects its intended function. Therefore, in our approach, we use the names of the elements that relate to a change to extract the keywords describing the change. The change-related elements include the name of the element that has been added/deleted in the code and the names of its parent or its sub-elements. Column 2 of Table I presents the elements that we consider as change-related for each type of changed element. For example, if a new class is added, then we consider the name of the class and the names of the methods and fields it contains.

As naming conventions differ from one language to another, the approach for extracting the keywords depends on the used language. For a concrete example, we consider the camelCase convention, which is used in several programming languages (e.g. Java and .NET). For the camelCase convention, the names are split according to the position of capital letters in the name, so that camelCase is split into ”camel” and ”case”. If several subsequent capital letters appear in the name like generateHTMLReport, then these letters are considered as one keyword so that the results is ”generate”, ”html” and ”report”. If a keyword is composed of one letter only or is a special character (not alphanumeric) then it is deleted.

2) Call Hierarchy: When a new element is added, considering only the names of the element and its parent/children might not be enough to determine the context of the change. Therefore, we need more information about when such an element is used. To get such information, we look at the call hierarchy of the element. We consider the call hierarchy when the added/removed element is a method or a class. By call hierarchy we mean all the methods/classes that are invoking the considered method or those invoking the a constructor of the considered class. The invocation can be either direct or via other methods/classes. As the call hierarchy of an element can be large, we should not go back too far in it. We expect that going back by one, two or three levels in the call hierarchy should be enough to gather relevant information about the context of the change.

To extract keywords from the elements identified in the call hierarchy, we use the same approach that we described in the previous section for extracting keywords from names of elements.

3) Documentation: The documentation of elements is also a valuable source of information that we consider in our approach. We add the terms in the documentation of important elements to our list of keywords. The third column of Table I presents which documentation we consider for each type of change.

4) Filtering and Grouping: Before tracing the keywords, we filter out irrelevant keywords and group changes together. Applying the keywords extraction approach is likely to generate many irrelevant keywords such as keywords relating to implementation details (set, get, string, etc.), very general keywords in the project (e.g. the name of the project), or stop words that might appear in the documentation of elements (a, the, that, etc.). Having irrelevant keywords in the list might have negative effect on the tracing of changes to requirements. Therefore it is important to filter out as many of these irrelevant keywords as possible. Filtering can be done in several ways. One possibility is to manually prepare a list that combines the most common words in the code and a stop word list. A more sophisticated way to construct the list is to build it automatically based on the frequency and appearance of keywords in the considered source code.

Considering each change separately will results in a very high number of changes, where each change might not be relevant on itself. Therefore it is important to group changes together. One possible way to group changes is to consider all changes affecting a class as a single change, where the keywords extracted from all the changes in that class are grouped together. Although this grouping might not be the most efficient one, it should reduce the number of changes to be traced to a reasonable number for medium size projects.

C. Identifying Affected Requirements

The last step of our approach is meant to identify the requirements that are affected by the change based on the keywords we extracted and grouped in the previous steps. This can be done by using IR-based techniques to trace the extracted keywords to requirements. As most IR-based tracing techniques perform similarly [8], the selection of a concrete technique to use is not such important. The requirements identified by the tracing tool are the ones that are likely to be affected by the change and thus are the ones that the maintainer should review.

IV. EVALUATION

This section describes the evaluation of our approach in a case study. Section IV-A describes the prototypical tool

we implemented to automate our approach. Section IV-B then describes the case study system, the iTrust Health care project.

The evaluation consists of two studies designed to answer the following research questions. In the first part, described in Section IV-C, we evaluate the effectiveness of our approach for identifying requirements-related changes (relevant changes), thus validating the first step of our approach as shown in Figure 1. The evaluation question is

> EQ1: How effective are the proposed heuristics for differentiating between changes which impact requirements and those which do not?

In the second part, described in Section IV-D, we evaluate the effectiveness of using change-related keywords for tracing the changes to requirements instead of tracing classes directly, thus validating the second and third step of our approach as shown in Figure 1. The evaluation question is

> EQ2: Does our approach of change-based tracing give better results than the class-based tracing? If yes, how much?

The metrics to assess the quality of the change identification (EQ1) and the quality of trace results (EQ2) are presented in sections IV-C1 and IV-D1.

Finally, in Section IV-E we discuss the significance of our findings and threats to validity. We do not combine the two parts in an end-to-end validation because there are no existing approaches to meaningfully compare to, as discussed in more detail in Section IV-E3.

A. Tools

In this section we present the two tools we used to run our experiment. The first tool is a prototype that we developed based on the first and second steps of our approach. This tool was used to compare and identify the relevant changes between two versions of source code and to extract the keywords related to these changes. The second tool is an information retrieval based traceability tool called RETRO. We used RETRO to trace the keywords found by our prototype to the requirements specification.

Prototype: Our prototype (1) compares two versions of code and identifies the requirements-related changes and (2) extracts the keywords related to each of the changes. Although the comparing technique and the keywords extraction parts are not completely separated from each other in the implementation, we present each part separately for the sake of comprehensibility.

The comparing part was developed based on an existing Java library for comparing Java API called JDiff [9]. We have chosen JDiff, as it has similarities with our comparing technique: JDiff detects the elements that are added or removed in the code, but does not detect changes in methods bodies. We adapted JDiff so that it ignores changes that are not relevant in our case, such as changes in elements signature. We also changed the default behaviour so that it also detects changes in private elements. To filter out rename operations, we implemented a comparator that compares element names and their call hierarchy: if an added element and a deleted one have similar names and/or have the same call hierarchy then the change is considered to be a rename and is ignored by the tool. The name similarity is computed based on the Levenshtein distance [7].

For each of the changes identified by the comparing part, the tool extracts a set of keywords related to it as presented in Section III-B. The tool contains a configurable list of keywords (stop word, project-specific common words) that are filtered out when building the list of keywords for each change.

The extracted keywords can then be reviewed by the user. There are two possible display configurations: either the keywords are grouped by change, which is very fine-granular, or grouped by class.

RETRO: RETRO (REquirements TRacing On target)[10] is an automated tool for generating traceability links among textual artefacts. RETRO implements various IR-based techniques for link generation. RETRO takes as input two lists of textual files: the high-level documents and the low-level documents and traces them to each other. The output of RETRO is a list of candidate links that are sorted according to their relevance.

RETRO includes other functionalities, which we did not use in our experiment, such as filtering links having relevance lower than certain threshold values, and entering analyst feedback to improve the generated links.

Configuration: In this paragraph we specify the configuration we used in our experiment for each of the tools. For our prototype, we considered changes in both public and private elements, and we set the depth of the call hierarchy to two. We also used the keywords grouped by class. For the RETRO tool, we used the default tracing method, which is the vector space retrieval with tf-idf (term frequency - inverse document frequency) term weighting.

B. Case Study

We used the iTrust Medical care project [11] as a case study for the evaluation. iTrust, is a tool for managing medical data and has been developed for teaching purposes at the North Carolina State University. The tool has a wiki-based requirements specification that includes functional requirements, non-functional requirements, a glossary, a set of global constrains (e.g. programming language, coding standards, etc.) and a section dedicated for specifying the data format for the input fields [12]. The tool is a web application that is developed using a combination of Java code and Java Server Page. A new version of the code, which is maintained by students in software engineering, is released every semester.

For the case study, we only considered the functional requirements, which are specified in the form of fine-grained

UC1 Create and Disable Patients Use Case

1.1 Preconditions: The iTrust HCP has authenticated himself or herself in the iTrust Medical Records system [UC3].

1.2 Main Flow: An HCP creates patients [S1] and disables patients [S2]. The create/disable patients and HCP transaction is logged [UC5].

1.3 Sub-flows:

[S1] The HCP enters a patient as a new user of iTrust Medical Records system. Only the name and email are is provided. An email with the patient's assigned MID and a secret key (the initial password) is personally provided to the user, with which the user can reset his/her password. The HCP can edit the patient according to data format 6.4 [E1] with all initial values (except patient MID) (...)

[S2] The HCP provides the MID of a patient for whom he/she wants to disable [E2]. The HCP provides a deceased date (data format 6.4). An optional diagnosis code is entered as the cause of death.

1.4 Alternative Flows:

[E1] The system prompts the enterer/editor to correct the format of a required data field because the input of that data field does not match that specified in data format 6.4 for patients.

[E2] (...)

Figure 2. Example Use Case from iTrust Requirements Specification [12], version of September 3rd, 2010

use cases. There are around 40 use cases in total. Figure 2 shows an example use case of the system.

For the code, we only considered the part written in Java as our prototype only works on Java code. We used versions 10 (release date: August 18th, 2010) and 11 (release date: January 7th, 2011) of the source code. To obtain the requirements that correspond as much as possible to each of these releases, we choose a wiki version from a date that is after the code release and before the beginning of the following semester. The reason is that, after the release, the project owners do a cleanup and maintenance for the requirements based on the work done by the students. Therefore we consider the requirements specification as of September 3rd, 2010 (the "old requirements") for the source code version 10 (the "old code version") and the requirements as of February 7th, 2011 (the "new requirements") for the source code version 11 (the "new code version") [12].

We manually compared the two versions of the code and identified the main changes that relate to the external behaviour of the system. To make sure that we did not miss any important change, we used the Java source compare in Eclipse, which identifies all textual changes (including addition/removal of spaces). Eclipse identified 91 changed classes.

We went through all of these classes and identified 14 different requirements-related changes, i.e. changes that should affect requirements. One requirement-related change can be scattered over several classes, so that each class contains a part of this change. The total number of classes that contain requirements-related changes is 31. Then we went through the requirements specification and identified all the use cases that are affected by each of these changes.

To check the completeness of our change list, we compared the old and the new versions of the requirements specification and looked for the requirements changed by the owners of the project. This comparison was challenging because of two reasons. First, we observed that both versions of the requirements specification do not perfectly match the respective code versions. Sometimes, the requirements specification listed a requirement that was not yet implemented in the respective code version. In other cases, the requirements specification was indeed outdated, so it did not reflect a recent behaviour-changing change in the respective code yet. Second, as we only consider the Java part of the source code, it is likely that we missed the changes that affect the jsp part only. This comparison was still helpful, as it helped us decide which uses cases should be updated for certain changes.

C. Study 1: Identification of Requirements-Related Changes (EQ1)

The goal of the first part is to evaluate how well the change identification step of our approach identifies requirements-related changes.

1) Experiment Design: We run our change identification step on the two iTrust source code versions. It reports a set of classes that are supposed to contain parts of the requirements-related changes.

To assess the performance of our approach, we measure its precision and recall. For the recall, we determine how many requirements-related changes are covered by the classes reported by our change identification step. Due to the scattering and tangling between requirements and code, a change in one requirement is likely to show up in several classes and methods in the code. For identifying whether a requirement is outdated, it is enough if one of the changed classes is reported. Therefore, a requirements-related change can be deemed covered if at least one class that contains a part of this change is reported. The recall measure is defined as the fraction of requirements-related changes covered by the retrieved classes.

For the precision, we determine how many of the retrieved classes are relevant for the requirements-related changes. The precision measure is defined as the fraction of retrieved classes that actually contain at least one part of a requirements-related change.

2) Results: Using our comparing tool, 33 classes were identified, covering 12 of the 14 requirements-related changes. Among the 33 identified classes, 26 actually contained parts of the 14 changes. The other 7 classes were simple refactorings. Thus, our approach achieved a precision of 26 / 33 = 79% and a recall of 12 / 14 = 85.7%.

To conclude the first study, we observe that our approach was able to find most requirements-related changes and to exclude the majority of irrelevant classes.

D. Study 2: Keyword Extraction and Tracing Results (EQ2)

In this part we evaluate how well our approach can identify affected requirements, i.e. use cases in this case study, based on the extracted keywords.

1) Experiment Design: We run the keyword extraction and tracing step for each of the 26 relevant classes from the previous part of the evaluation. Our choice for tracing the relevant classes only and not all of the 33 changed classes is explained in Section IV-E3. The output of the two steps is a ranked list of candidate use cases for each class which are suggested to be related to the requirements-related changes in this class.

To assess the quality of the rankings, we compare them to the true relation between the changes and the use cases, which we defined manually as discussed in Section IV-B. Each class we create a ranking for is related to one of the 14 requirements-related changes. Thus, our approach should report the use cases affected by that requirements-related change. The first two columns of Table II show this true relation of classes to use cases for the 14 requirements-related changes, which forms the ground truth for the keyword extraction and tracing step. For two classes, namely the `ActivityFeedAction` and the `ViewHelperAction`, a new requirement was introduced, so no use case could be matched in the requirements specification (marked NEW in Table II). Let Z denote the set of considered classes and let U denoted the set of related use cases. We denote the true relation as $T \subset Z \times U$. A true link between a class c and a use case u is denoted $t = (c, u) \in T$.

To assess how well our approach performs compared to existing approaches, we compare our approach to a simple class-based tracing approach between the classes and the use cases of the requirements specification as a baseline, described in the following. In the class-based tracing, we use all the keywords in the file of the class as input to compute the similarity of that class with the use cases. The output of the class-based tracing is a ranked list of candidate use cases that are suggested to be related to the class in general. Thus, to compare the usefulness of our change-based approach and the class-based tracing in the requirements update scenario, we compare how close the rankings produced by the two approaches are to the true relation described above.

The produced rankings do not necessarily rank all use cases, but only a subset. For statistical validity, we use a fractional ranking where tied use cases receive a fractional rank number that is the mean of the ranking positions they would receive in ordinal ranking. Let us denote a suggested ranking R_c of a subset of the use cases $U_{R_c} \subset U$ for a class c as a function $R_c : U_{R_c} \to \mathbb{R}$ so that the rank of a use case u is given by $R_c(u)$. Furthermore, let $R_Z = \{R_c \,|c \in Z\}$ denote the set of rankings suggested by an approach for all classes.

To measure the quality of the rankings, we use three measures, namely the the median rank, precision, and recall.

First, we measure the median rank of the true links in the rankings produced by each approach. For each class c, let the use cases that should be found be denoted $U_c = \{u \in U \,|(c, u) \in T\}$. Then, the median rank of the true links is $\tilde{R}_c(u)$ for $c \in Z, u \in U_c \cap U_{R_c}$.

Note that this calculation ignores situations where a true link is not contained in a ranking, i.e. where $U_c \cap U_{R_c} = \emptyset$. As we will see later, this calculation favours the class-based tracing approach, so we do not present a more complex median calculation that accounts for missing ranks by e.g. assigning a default rank at the end of the ranking.

Second, we measure precision and recall. The precision of an approach is the fraction of retrieved true links, and the recall is the fraction of true links that were retrieved. Because it is easier for developers if true links are suggested early in a ranking, we study the precision and recall at a cut-off rank n, i.e. only links retrieved at ranks lower than n are considered (cf. [13, Sec. 4.9.3]). More formally, the true links suggested by an approach up to cut-off rank n is

$$cutTrue(R_Z, T, n) = \{(c, u) \,|(c, u) \in T, R_c \in R_Z, R_c(u) \leq n\}$$

The number of all links suggested by an approach up to rank n is

$$cutAll(R_Z, T, n) = \{(c, u) \,|R_c \in R_Z, u \in U_{R_c}, R_c(u) \leq n\}$$

Then, the precision at n is

$$precision(R_Z, T, n) = \frac{|cutTrue(R_Z, T, n)|}{|cutAll(R_Z, T, n)|}$$

and the recall at n is

$$recall(R_Z, T, n) = \frac{|cutTrue(R_Z, T, n)|}{|T|}$$

To study how early relevant links are suggested by the approach, we increase the cut-off rank n from 1 to half the number of use cases to create a precision-recall graph [13, Sec. 4.9.3].

2) Results: Table II shows the resulting ranks of the correct use cases as produced by our change-based approach and the comparison class-based approach. In three cases, the true link was not retrieved at all by an approach, which is marked by NA in Table II. We observe that our tool performs better in 16 of 26 cases and even considerably better (i.e. 5 or more ranks better) in 7 cases. The class tracing performs better in only 4 cases, two of which are considerably better. In 5 cases, the approaches perform equally well.

When considering the cases in which our approach performed considerably worse, we detect two problems. For use case 23 for class `ApptDAO`, we find out that the problem comes from the fact the developers used abbreviations (“appt” for “appointment”), therefore our tool could not trace it to the appointment use case. However, class-based tracing was better because there was a message that should be displayed to the user in the body of the method and which

Table II
RANKS FOR CLASSES. A CELL IS MARKED GREY IF ONE OF ITS RANKS IS MORE THAN 5 RANKS BETTER THAN THE RANK OF THE OTHER APPROACH. A RANK IS UNDERLINED IF IT IS BETWEEN ONE AND FOUR RANKS BETTER THAN THE RANK OF THE OTHER APPROACH.

Id	Class	Affected Use Cases	Change-based Tracing	Standard Class-Based Tracing
1	ActivityFeedAction	NEW		
2	AddRemoteMonitoringDataAction	UC34	1	2
3	ApptDAO	UC22	29	1
4	DAOFactory	UC15, UC4	5, 30	21, 34
5	EditApptAction	UC22	3	3
6	EditOfficeVisitForm	UC11	4	1
7	EventLoggingAction	UC5	1	3
8	HealthData	UC10	7	14
9	LoginFailureAction	UC3	1	1
10	OfficeVisitDAO	UC11	4	1
11	OverrideReasonBean	UC15	28	30
12	OverrideReasonBeanLoader	UC15	11	29
13	OverrideReasonBeanValidator	UC15	23	34
14	PrescriptionBean	UC37, UC15	21, 1	16, 12
15	ProfilePhotoAction	UC4	7	8
16	ProfilePhotoDAO	UC4	33	37
17	ProfilePhotoServlet	UC4	NA	NA
18	ReasonCodesDAO	UC15	3	7
19	RemoteMonitoringDAO	UC34	1	1
20	RemoteMonitoringDataBean	UC34	1	5
21	RemoteMonitoringListsBeanLoader	UC34	1	26
22	TelemedicineBean	UC34	1	NA
23	TransactionDAO	UC5	2	4
24	UpdateReasonCodeListAction	UC15	1	1
25	ViewHelperAction	NEW		
26	ViewMyRemoteMonitoringListAction	UC34	1	7

contains the keyword appointment in it. For use case 37 and class `PrescriptionBean`, we observe that the addition of an ORC (an overriding reason code) was better ranked by the class-based tracing than by our approach (16 to 21). We found that the bean was only called from jsp classes, which were not considered in this study. The lack of a call hierarchy might have hindered our approach in this case. However, note that both approaches did not perform well and produced a high rank.

The median rank of the correct use case in our approach is $\tilde{x} = 4$, while it is $\tilde{y} = 7$ for the class-based approach. In fact, our approach produces lower (i.e., better) ranks at a significance level of 0.05. We use a one-tailed Wilcoxon Signed Rank test with continuity correction as performed by the R statistics tool [14] because it is applicable to ordinal scales and we assume that the differences $x - y$ are independently distributed [15]. Our null hypothesis H_0 is that the difference in ranks $x - y$ is symmetric about 0 or larger. H_0 is rejected with $p = 0.013$.

Figure 3 shows the results for precision and recall at cut-off ranks n for $1 \leq n \leq 15$. We observe that our approach performs better with respect to both precision and recall. For example, if only the first first returned use case is considered ($n = 1$), the class tracing has a precision of 0.23 and a recall of 0.23 while our approach has a precision 0.38 and a recall

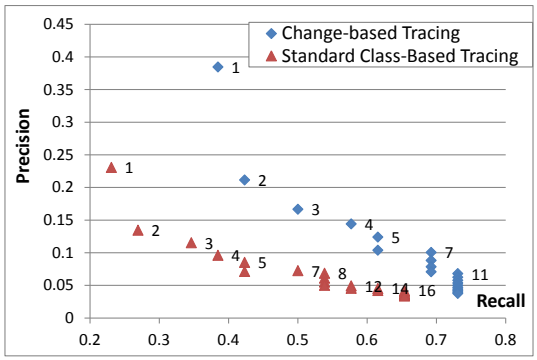

Figure 3. Precision / recall at n for ranks smaller than 15, the respective cut-off rank n is annotated to the data points.

of 0.38, which is an improvement of 66% [2]. Furthermore, for a fixed recall value, the precision of our approach is at least twice as good as for the class-based approach.

To summarize, we observe that our approach was able to find better results than the class-based approach. Thus, for the second study we conclude that in this case study, a hint generation for outdated requirements based on keywords extracted from the change and its context is more useful than using class-based tracing for identifying outdated requirements.

E. Discussion

In this section we discuss the significance of our findings (Section IV-E1), the threats to validity of our study (Section IV-E2), and argue for the chosen two-part validation strategy (Section IV-E3).

1) Approach Evaluation: The results obtained from both parts of the evaluation give a positive indicator about the relevance of the approach. In fact, our approach succeeded to cover most of the important changes and gave results that are in most of the cases better than those obtained by using class-based tracing. Additionally, the approach does not require much effort, as it can either be run fully automatically or with some user feedback. In the current evaluation, the user feedback consists in removing the 7 classes that contained simple refactoring and which were still identified by the tool. An important characteristic of our approach is that it filters out much of the irrelevant information that hinder the tracing (e.g. the import packages in a class). It also considers the context of the change through the call hierarchy of elements. Another plus of our approach is that it is configurable (e.g. depth of call hierarchy, elements to be considered, etc.) thus it can be adapted to the characteristics of the used project.

The two main limitations of the approach are that (1) it can miss some relevant changes and (2) it does not generate links that are 100% correct. The first problem is related to the

[2]Note that precision and recall coincidently have the same value, as the number of classes is 26 and the number of true links is 26, too.

compromise between identifying as many relevant changes as possible, and identifying relevant changes only. If the approach is extended so that it covers more changes (e.g. changes in methods bodies) then it will detect more relevant and more irrelevant ones. The second limitation is a normal traceability problem: tracing is based on a pure textual analysis that only considers the keywords appearing in the traced documents without considering the intent behind it. Therefore, such a technique generates many false links between documents that are not related but contain similar keywords and misses the links between related documents that do not contain similar keywords.

Despite these limitations, the approach can still be very useful as it can support the maintainer during the update by suggesting him the requirements that are likely to be affected. Instead of manually analyzing all requirements for whether they need to be updated, the maintainer can focus on the requirements suggested by our approach first, which we expect to decrease the effort for updating the requirements considerably. To quantitatively assess how useful such tool is, we intend to conduct an experiment where we compare how efficient the update of requirements is with the help of our approach compared to manual requirements updated and standard-traceability-based requirements update.

2) Threats to Validity:

External Validity: As the evaluation was done based on one project only, the findings cannot be generalised to other types of projects. In fact, the results depend on several projects parameters such as the coding style, the structure and types of the requirements specification, etc. Nevertheless, the positive results obtained in the current evaluation indicate that the approach is beneficial for at least one type of software projects. Further evaluations will be conducted in the future, to assess the effectiveness of our approach on other types of projects.

Internal validity: To ensure internal validity, we need to check whether the superiority of the results obtained by our approach over the class-based tracing approach is due to our approach. As we ran both experiments using the same project (same requirements specification) and the same traceability tool, the only parameter that has changed is the use of the change context instead of using the whole class. Therefore, the improvement in results can only be due to our approach of extracting keywords from the change context.

A second threat to internal validity is related to the identification of the affected requirements (the ground truth). Identifying which requirements should be updated after a change can be a subjective matter. To avoid any bias, we did the manual identification of affected requirements before running any of the experiments.

3) Validation Strategy: We preferred an evaluation in two studies rather over doing an end-to-end validation of the approach, because we did not find other approaches performing the same task as ours that we could use to compare our approach to. We think that comparing our use of keywords extracted from the change context to class-based tracing in the second study is more relevant than presenting raw values for the whole approach that are not comparable to anything. Furthermore, comparing the whole approach to a class-based tracing approach would be unfair: We would need to trace all of the changed classes (91 classes) for the class-based approach, which would result in a very low the precision for the class-based approach because many of these classes were only refactored. This motivates our choice for tracing only the 26 relevant classes identified in Section IV-C2 with both approaches.

V. Related Work

There are two categories of existing approaches to requirements update, namely (1) approaches prescribing to update requirements first and propagate the change to the code and (2) trace generation approaches generating traces between requirements specification and code.

First, the approaches for updating requirements specifications assume an ideal maintenance process where first the requirements are updated then the changes are propagated to the source code [5] [16]. In our approach, however, we consider the frequently recurrent case where only the code was updated while other documents, including requirements, were not modified. Here, our approach uses the changes that were done at the code level to support the update of requirements.

In a previous work [17], we propose a test-based approach for identifying requirements affected by change. A set of high-level tests and traceability links between these tests and requirements are used to identify the requirements impacted by each implemented change. In contrast, in our current work we do the analysis on the source code directly, so there is no need for the high-level tests and no need for any traceability links.

The main approach that is used for propagating changes between software artefacts, and which is also applicable for propagating changes between source code and requirements, is software traceability. Approaches for automatically generating traceability links between software artifacts and for using traceability to manage and propagate change do exist [6] [18] [19]. Our current work can be considered as a special traceability approach that focuses on tracing only the relevant code changes to requirements. There are two main differences between our approach and existing traceability approaches. First, our approach includes a feature for identifying the relevant changes that should be traced in the code. Second, we propose a new way for selecting the set of terms to be traced.

VI. Future Work

There are two main parts for the future work. One part is about extending and improving the current approach. The other part is about the evaluation of the approach.

A. Extending the Approach

There are two techniques that we intend to incorporate in our approach. First, we plan to introduce weights for the keywords depending on their source. For example, keywords extracted directly from the change could be assigned a higher weight than keywords extracted from containing classes or from the call hierarchy.

Second, we plan to use the requirements specification to further improve the weighting of the extracted keywords. This can be done by considering the occurrence of keywords in the requirements specification. We will use what we call *keyword specificity*, which we define as follows: if one keyword appears in several scattered parts of the requirements specification then it is not very specific and will either have a low weight or be filtered out completely.

B. Evaluating the Approach

To further evaluate our approach, we plan to apply it to different types of projects in the future. The goal of the evaluation is to find out for which type of projects our approach works best. Another part of the evaluation will be about the usefulness of our approach for requirements update. We will explore, using a controlled experiment, the effect of our approach on the maintainer's efficiency during the update and on the quality of the update.

VII. Conclusion

In this work we present a new approach for identifying outdated requirements based on an analysis of code changes. Our approach has three main steps: First, the new and old versions of source code are compared and relevant changes are detected. Then a set of keywords describing the change is extracted from the names of the elements related to the change, their documentation and their call hierarchy. Finally related requirements are identified by tracing the extracted keywords to the requirements specification. Our approach is meant to support the requirements update task when no predefined traceability links are available. Compared to a class-based tracing approach, our approach yields better results in terms of precision, recall, and the ranking of true links. The next step of our work would be to evaluate how useful our approach is for supporting the requirements update task.

Acknowledgements

We thank Simon Käser for implementing the prototype used for the evaluation. We also would like to thank Laurie Williams and Ben Smith for providing us with information about the iTrust project. This work is funded by the Swiss National Science Foundation under grant PDFMP2-122969.

References

[1] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proc Int. Conf. Design of communication: documenting & designing for pervasive information*, ser. SIGDOC '05, 2005, pp. 68–75.

[2] K. H. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *ICSE - Future of SE Track*, 2000, pp. 73–87.

[3] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Softw.*, vol. 20, no. 6, pp. 35–39, 2003.

[4] T. Gorschek and M. Svahnberg, "Requirements experience in practice: Studies of six companies," in *Engineering and Managing Software Requirements*, A. Aurum and C. Wohlin, Eds. Springer Verlag, 2005, pp. 405–424.

[5] A. Herrmann, A. Wallnöfer, and B. Paech, "Specifying changes only–a case study on delta requirements," in *Requirements Engineering: Foundation for Software Quality*, 2009, pp. 45–58.

[6] J. Hayes, A. Dekhtyar, and S. Sundaram, "Advancing candidate link generation for requirements tracing: the study of methods," *IEEE Trans. Softw. Eng.*, vol. 32, no. 1, pp. 4–19, Jan. 2006.

[7] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[8] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *Proc. Int. Conf. Program Comprehension*, 2010, pp. 68–71.

[9] M. B. Doar, "JDiff – what really changed?" *Java Developer's Journal*, 2002.

[10] J. Hayes, A. Dekhtyar, S. Sundaram, E. Holbrook, S. Vadlamudi, and A. April, "REquirements TRacing On target (RETRO): improving software maintenance through traceability recovery," *Innovations in Systems and Software Engineering*, vol. 3, no. 3, pp. 193–202, 2007.

[11] A. Meneely, B. Smith, and L. Williams, "Appendix B: iTrust electronic health care system case study," in *Software and Systems Traceability*. Springer Verlag, 2012.

[12] L. Williams, T. Xie, A. Meneely, L. Hayward, and J. King, "iTrust medical care requirements specification," Versions of September 3rd, 2010: agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=requirements& rev=1283530873 and of February 7th, 2011: rev=1297120633.

[13] S. Dominich, *The Modern Algebra of Information Retrieval*. Springer Verlag, 2008, vol. 24.

[14] R Development Core Team, *R: A Language and Environment for Statistical Computing*, 2010, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org/

[15] R. Lowry, "Concepts and applications of inferential statistics, chapter 12a: The Wilcoxon signed-rank test," 2012. [Online]. Available: faculty.vassar.edu/lowry/ch12a.html

[16] N. Ernst, A. Borgida, and J. Mylopoulos, "Requirements evolution drives software evolution," in *Proc. Joint ERCIM Workshop Software Evolution (EVOL) and Int. Workshop Principles of Software Evolution (IWPSE)*, 2011, pp. 16–20.

[17] E. Ben Charrada and M. Glinz, "An automated hint generation approach for supporting the evolution of requirements specifications," in *Proc. Joint ERCIM Workshop Software Evolution (EVOL) and Int. Workshop Principles of Software Evolution (IWPSE)*, 2010, pp. 58–62.

[18] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, 2002.

[19] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-based traceability for managing evolutionary change," *IEEE Trans. Softw. Eng.*, vol. 29, no. 9, pp. 796–810, 2003.